ELSEVIER
Science of Computer Programming 45 (2002) 155–174
Science of Computer Programming
www.elsevier.com/locate/scico

A change impact model for changeability assessment in object-oriented software systems ☆

M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, François Lustman*

Département IRO, Université de Montréal C.P. 6128, succursale Centre-ville, Montréal, Québec H3C 3J7, Canada

Abstract

Growing maintenance costs have become a major concern for developers and users of software systems. Changeability is an important aspect of maintainability, especially in environments where software changes are frequently required. In this work, the assumption that high-level design has an influence on maintainability is carried over to changeability and investigated for that quality characteristics. The approach taken to assess the changeability of an object-oriented (OO) system is to compute the impact of changes made to classes of the system. A change impact model is defined at the conceptual level and mapped on the C++ language. In order to experiment the model as a changeability indicator on large industrial software systems, an experiment involving the impact of one change is carried out on a telecommunications system. The results suggest that the software can easily absorb that kind of change and that well chosen conventional OO design metrics can be used as indicators of changeability.

Keywords: Software engineering; Software quality; Maintenance; Change impact; Design; Software metrics; C++ language

1. Introduction

In the quality standard ISO 9126, maintainability is one out of six quality characteristics [14]. Over the years, cumulative data have shown that maintenance is also a major

☆ This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada).

* Corresponding author.
E-mail addresses: ajmal.chaumun@cgi.ca (M.A. Chaumun), kabaili@iro.umontreal.ca (H. Kabaili), keller@iro.umontreal.ca (R.K. Keller), lustman@iro.umontreal.ca (F. Lustman).

0167-6423/02/$ - see front matter
PII: S0167-6423(02)00058-8

cost concern, as a matter of fact, a growing cost concern [24]. While many reasons are forwarded in an attempt to explain the spiraling costs of software maintenance, a consensus has emerged that the maintainability of a software system is dependent on its design [25], in the procedural paradigm as well as in the OO paradigm.

In ISO 9126, maintainability has four components, analyzability, test ability, stability and changeability. In some application areas like telecommunications, software systems are evolving constantly. Moreover, there are organizations which do not develop the software they operate, but purchase it. They are not directly interested in testability or diagnostic, but in the software's ability to sustain an on-going flow of changes. In this work, the focus will be on that single aspect of maintainability, i.e., changeability. In the SPOOL project, a joint project with Bell Canada, we are investigating the dependency between the changeability of software systems and their design.

One way of assessing changeability is to assess the impact of changes (change impact analysis). Our research approach is both analytical and experimental. It involves defining a change impact model, more complete and more general than those presented in the literature, and applying it on an industrial software system to assess its changeability. Work related to maintainability and changeability is presented in Section 2. The change impact model, its application to C++, and its prototype implementation are described in Section 3. A case study for testing the model and its implementation was carried out on a medium-size C++ industrial software system, and is presented in Section 4. The results of our work are discussed in the Conclusion, where further work is also outlined.

2. Related work

Design characterization is mostly done through metrics. A conventional distinction is made between architectural or high-level design and algorithmic or low-level design, and according to Rombach, the former has more influence on maintainability than the latter [25]. In the realm of OO design, numerous design metrics have been published [1,9,22]. One suite of OO design metrics has been proposed by Chidamber and Kemerer and progressively refined [8,9,10]. The suite (called C&K metrics later in this article), theoretically well-grounded, comprises four inter-class metrics, DIT (depth of inheritance tree), NOC (number of children), CBO (coupling between objects), and RFC (response for a class), and two intra-class metrics, WMC (weighted methods per class), and LCOM (lack of cohesion in methods). Several studies were conducted to validate the metrics and to relate them to some maintainability property. Li and Henry took five of the above metrics (CBO was excluded), added three of their own (DAC: number of ADTs defined in a class, MPC: message-passing coupling, and NOM: number of methods), and tested that set on two commercial OO systems numbering 39 and 70 classes [18]. They were able to conclude that there is a strong relationship between the metrics and maintenance effort (expressed in number of lines changed). Later on, they restricted themselves to those metrics available from design documents, and were able to draw the same conclusions [19]. Abreu in [2] defined a set of metrics called MOOD metrics. Each of those metrics refers to a basic structural mechanism of

the object-oriented paradigm as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphism (PF) and message-passing (CF), and are expressed as quotients. Basili et al. were interested in a specific part of maintenance, i.e., fault detection and fault proneness [5]. Experiments on eight systems developed by students, showed that individually, the C&K metrics were related to the probability of fault detection and that globally they were also good indicators of faulty modules.

Some studies on the relationship between design and maintainability were based on other design metrics. Hsia et al., for example, studied the effect of architecture on maintainability [13]. On two systems designed by students, they measured maintainability (adding new features) and its relationship to architecture, namely the broadness of the inheritance trees. It turned out that maintainability is better for systems with broader trees, i.e., shallower inheritance trees. Briand et al. defined 18 coupling measures between classes and studied their significance in predicting fault-proneness in several industrial systems on which they had gathered maintenance data [6]. They were able to conclude that some of the coupling metrics were significant predictors of fault-proneness.

Less work has been conducted on the matter of change impact. Han developed an approach for computing the change impact on design and implementation documents [12]. Artifact dependencies involve inheritance, aggregation and association. Change impact is identified based on the value of a Boolean expression. However, software changeability is not really assessed. Lindvall identified the most common and frequent C++ changes so that change models can be built to help developers make predictions regarding future requirements [21]. In [4] the authors predicted the size of evolving OO systems based on the analysis of the classes impacted by a change request. They predicted the size of changes in terms of added/modified lines of code. Kiran et al. compared the maintainability of software systems in the functional paradigm and in the OO paradigm [15]. They used programs developed by students and defined sets of changes which were implemented by graduate students. Results suggest that the maintenance effort is less important in the OO paradigm than in the functional paradigm. In particular, the impact of the set of changes considered is more localized in the OO paradigm than in the functional paradigm. Kung et al. were interested in the system-wide impact of changes for regression-testing purposes. They defined a classification of changes and impacts resulting from the changes based on the three links inheritance, association, and aggregation. They also defined formal algorithms to calculate all the impacted classes including ripple effects [16,17]. Li and Offutt proposed also algorithms for calculating the complete impact of changes made in a given class [20]. They were interested in the effects of encapsulation, inheritance, and polymorphism, on the impact.

This short literature survey can be summarized as follows. Most of the results presented above are derived from the study of small commercial systems or even of systems developed in course assignments. Based on these experiments, a growing body of evidence suggests that design has an influence on maintainability, and that the C&K metrics, for example, may be considered as maintainability indicators. On the other hand, changeability has been less studied than test detection or the overall maintenance effort. In particular, there is no evidence that design has an influence on changeability. Moreover, most of the change impact studies propose incomplete models. Kiran

et al. considered only inheritance, aggregation and association but not invocation and friendship [15]. Li and Offutt did not consider changes in inheritance links, nor virtual methods [20]. The association and aggregation links were not fully covered in their impact calculation algorithms either. Kung et al. did not consider the impact of data change and of method change because it had already been covered by others [17].

In summary, most results on the influence of design on changeability come out of small systems, and the change impact models we found in the literature are incomplete or not systematic

3. Change impact model

3.1. Objectives

When a change to a system is considered, it is necessary to identify the components of the system which will be impacted as a result of that change. This is to ensure that the system will still run correctly after the change is implemented. Our concern is focussed on how the system reacts to that change and to other changes in general, i.e., how capable the system is to absorb changes. A system absorbs easily a change if the number of impacted components is low. We expect systems designed differently to be affected quite differently upon similar change requests. Therefore, we are interested in exploring how exactly the design of a system affects its changeability.

The design of a system can be pictured as a set of software artifacts (classes) interacting with one another. Inter-class links are assumed to have greater influence on maintainability than intra-class links [25]. We contend that this assumption applies particularly to changeability. Thus, we focus on how the various links between classes do in fact influence changeability.

Our goal in the SPOOL project is to define a list of changes and a change impact model as complete and systematic as possible. The model should be language-independent, i.e., be situated at the design level. Also, it should allow for the concise and systematic impact calculation by using a formal approach. Finally, it should be applicable on industrial strength software systems with hundreds or even thousands of classes.

3.2. Conceptual model

The object artifacts available in the design phase are classes and their relationships. A class is defined as a group of methods which serve as public interface or for internal operations, and a section of variables which define the state of instances of the class. A component refers to either a class, a method, or a variable.

3.2.1. Changes

We define a change to a system as one to any of the three components. Examples are the addition of a variable, change in a method's scope from public to protected or the removal of the relationship between a class and its parent. The main changes to

OO systems at the design level are identified. They are categorized according to the component they affect and a total of 13 changes is identified:

(i) Addition, deletion, type change and scope change (variable component)
(ii) Addition, deletion, implementation change, signature change and scope change (method component)
(iii) Addition, deletion and structure change (class component)

3.2.2. Links

Once a given component is subject to change, we are interested in knowing which other parts (classes) in the rest of the system will be affected by this change. A specific part may be affected, in case it is 'connected' to the changed component via some link(s) between them. These links are of one of the following four types: association (**S**), aggregation (**G**), inheritance (**H**), and invocation (**I**). We also consider a special notation commonly used in the Boolean algebra; The absence of an operator between 2 links means an intersection. The "+" operator means an union. The "∼" before a link means the negation; the set of classes not associated with that special link. For example ∼**G** means the set of classes that are not linked to the specified class by the aggregation link.

Different OO modeling techniques have given slightly different definitions of the term association. In this work, we use the definition given in UML [26]. Aggregation is a form of association that specifies a whole-part relationship between two classes. Inheritance between two classes means that the derived class can benefit from whatever has already been defined in the base class. When methods defined in one class are being invoked by methods in another class, this is referred to as invocation. The links are independent from each other, and we can expect to find any number and type of links between two classes.

We also look for impact within the changed class itself. For example, a change in the type of a variable of one class leads to an impact in all the methods in the same class which use that variable. We introduce an artificial link called "local" (***L***) to denote such impact.

3.2.3. Impact

The impact of a change depends on two main factors. For one thing, different types of change lead to different sets of impacted classes. For example, the change in the type of a variable has an impact in all the classes referencing this variable whereas the addition of a parent to a class may cause impact in at least all derived classes.

Given a type of change, the type of link between classes is the second main factor to influence the impact result. Consider a change in the scope of a method from public to protected. Classes which invoke this method will be impacted, except for those who are derived from the changed class. Note that more than one type of link between the changed class and an impacted class can be involved in the calculation.

Thus, for a given change ch_i in class cl_j, the set of impacted classes is expressed as a Boolean expression in which the variables stand for the links. For example, the

impact formula for such a hypothetical change may be given by

$$Impact\ (cl_j, ch_i) = \mathbf{S} \sim \mathbf{H} + \mathbf{G},$$

meaning that classes which are in association (**S**) with, and not derived (∼**H**) from the changed class cl_j, or classes which are in aggregation (**G**) with cl_j are impacted. As examples, the change impact formulae for a change to each component type are as follows:

(i) "Variable deletion" change:

$$Impact\ (cl_j, ch_i) = \mathbf{S} + \boldsymbol{L}.$$

(ii) "Method scope change from public to protected" change:

$$Impact\ (cl_j, ch_i) = \mathbf{I} \sim \mathbf{H}.$$

(iii) "Class deletion" change:

$$Impact\ (cl_j, ch_i) = \mathbf{H} + \mathbf{G} + \mathbf{S} + \mathbf{I}.$$

It is worth noting that we search only for direct impact, i.e., we search for impact only in classes which directly interact with the changed class.

3.3. Application to C++

Design documents, if available, are all too often not consistent and do not reflect the reality of the system. In fact, the only document we can be sure to truly correspond to the running system, is the source code. The industrial systems targeted for experimentation and provided by our project partner are written in the C++ language. For these reasons, we decided to map our conceptual model into the C++ language.

3.3.1. Changes

First of all, we came up with a list of possible changes in C++. The changes are categorized in three groups based on the component (class or method or variable) which is being subject to change. A change in C++ is thus defined as the result of the syntactic code transformation pertaining to an identified component. The following illustrate some changes:

(i) The code change from "class c2: *public* c1" to "class c2: *protected* c1" corresponds to a change in inheritance derivation for class *c2*.
(ii) The code change from "void m (*void*)" to "void m (*int a*)" represents a signature change in method *m*.
(iii) The code change from "*int* v;" to "*double* v;" represents a change in the type of variable *v*.

We extended the list of 13 changes identified at the design level to address the code level and the specificities of C++. For instance, we introduced the change "variable

change from *static* to *non-static*". Another example of a refined change is the "variable scope change" that may be sub-classified as six changes:

(i) From public to private
(ii) From public to protected
(iii) From protected to private
(iv) From protected to public
(v) From private to public
(vi) From private to protected.

The final list contains a total of 66 changes, comprising 12 changes for variable, 35 for method, and 19 for class—refer to Table 1.

We treat changes one at a time and calculate their impact likewise. For example, for the code transformation "class c2: *c1*, *c0* {... }" to "class c2 {... }", we say that *c2* has deleted parent *c1* followed by *c2* has deleted parent *c0* instead of *c2* has deleted parents *c1* and *c0*. We also consider the changes to be non-overlapping, i.e., a change to a method or variable component is not also a change to the class component that comprises the method or variable at hand.

3.3.2. Links

The next step is to establish which links are represented in C++ and how to identify them. The four links (**S**, **G**, **H** and **I**) in the conceptual model are encountered in C++. In addition, we include a fifth link, friendship (**F**), a link which does not exist at the design level but is an integral part of the language. Below, we illustrate these five links with examples. We consider *C1* as the class to be changed and look for potential impact in class *C2*, with *C2* being linked to *C1*. Comments in the source text are inserted using the double slash (//). Note that the 'local' link (***L***) is considered conceptual and is derived, when the changed class coincides with the impacted class.

(S) Association

```
class C2 {
    ...
    C2_m1( ) {
      // C1_v1 is a variable of C1.
      // C2_v1 is a variable of C2.
      // o1 is a global object or
      // global object reference of C1.
      ... C2_v1 = o1.C1_v1;
      // o1ptr is a global object pointer of C1.
      ... C2_v1 = o1ptr -> C1_v1;
    }
    ...
      // C1_v1 is used in the actual parameter list.
      C2_m2 (... ,o1.C1_v1, ... );
      C2_m3 (... ,o1ptr -> C1_v1, ... );
};
```

Table 1
Impact result for all changes (C++)

Change Id	Change description	Impact expression	Local impact
v.1.1	Variable value change	—	—
v.1.2	Variable type change	S	L
v.1.3	Variable addition	—	—
v.1.4	Variable deletion	S	L
v.1.5	Variable scope change		
v.1.5.1	Public -> Private	S∼F	—
v.1.5.2	Public -> Protected	S∼H∼F	—
v.1.5.3	Protected -> Private	SH∼F	—
v.1.5.4	Protected -> Public	—	—
v.1.5.5	Private -> Public	—	—
v.1.5.6	Private -> Protected	—	—
v.1.6	Variable change (*Static*/*Non-static*)		
v.1.6.1	Static -> Non-static	S	L
v.1.6.2	Non-static -> Static	—	—
m.2.1	Method change (*Static*/*Non-static*)		
m.2.1.1	Static -> Non-static	I	L
m.2.1.2	Non-static -> Static	—	L
m.2.2	Method change (*Virtual*/*Non-Virtual*/*Pure virtual*)		
m.2.2.1	Virtual -> Non-virtual	—	—
m.2.2.2	Non-virtual -> Virtual	—	—
m.2.2.3	Virtual -> Pure virtual	H+*ie*(3.1.1)	L
m.2.2.4	Non-virtual -> Pure virtual	H+*ie*(3.1.1)	L
m.2.2.5	Pure virtual -> Virtual	H+*ie*(3.1.2)	L
m.2.2.6	Pure virtual - Non-virtual	H+*ie*(3.1.2)	L
m.2.3	Method return type change		
m.2.3.1	Non pure virtual method	I+*ie*(3.1.2)	L
m.2.3.2	Pure virtual method	H	L
m.2.4	Method implementation change	—	L
m.2.5	Method signature change		
m.2.5.1	Non pure virtual method	I	L
m.2.5.2	Pure virtual method	H	L
m.2.6	Method scope change		
m.2.6.1	Public -> Private		
m.2.6.1.1	Non virtual method	I∼F	—
m.2.6.1.2	Virtual method	I∼F	—
m.2.6.1.3	Pure virtual method	—	—
m.2.6.2	Public -> Protected		
m.2.6.2.1	Non virtual method	∼HI∼F	—
m.2.6.2.2	Virtual method	∼HI∼F	—
m.2.6.2.3	Pure virtual method	—	—
m.2.6.3	Protected -> Private		
m.2.6.3.1	Non virtual method	HI∼F	—
m.2.6.3.2	Virtual method	HI∼F	—
m.2.6.3.3	Pure virtual method	—	—
m.2.6.4	Protected -> Public		

Table 1 (*continued*)

Change Id	Change description	Impact expression	Local impact
m.2.6.4.1	Non virtual method	—	—
m.2.6.4.2	Virtual method	—	—
m.2.6.4.3	Pure virtual method	—	—
m.2.6.5	Private -> Public		
m.2.6.5.1	Non virtual method	—	—
m.2.6.5.2	Virtual method	—	—
m.2.6.5.3	Pure virtual method	—	—
m.2.6.6	Private -> Protected		
m.2.6.6.1	Non virtual method	—	—
m.2.6.6.2	Virtual method	—	—
m.2.6.6.3	Pure virtual method	—	—
m.2.7	Method addition		
m.2.7.1	Pure virtual method	*ie*(3.1.1)	—
m.2.7.2	Virtual & non-virtal method	I+*ie*(3.1.2)	L
m.2.8	Method deletion		
m.2.8.1	Pure virtual method	*ie*(3.1.2)	—
m.2.8.2	Virtual & non-virtual method	I+*ie*(3.1.1)	L
c.3.1	Class change (*Abstract/Non-abstract*)		
c.3.1.1	Non-abstract -> Abstract	G+H+I	L
c.3.1.2	Abstract -> Non-abstract	H	L
c.3.2	Class friendship relation change		
c.3.2.1	Add friend	—	—
c.3.2.2	Delete friend	F(S+G+H+I)	—
c.3.3	Class deletion		
c.3.3.1	Non-abstract class	S+G+H+I	—
c.3.3.2	Abstract class	S+H+I	—
c.3.4	Class inheritance derivation		
c.3.4.1	Public -> Private	$\sim$F(S + I)	—
c.3.4.2	Public -> Protected	$\sim$H$\sim$F(S + I)	—
c.3.4.3	Protected -> Private	H$\sim$F (S + $\sim$SG + $\sim$SI)	—
c.3.4.4	Protected -> Public	—	—
c.3.4.5	Private -> Public	—	—
c.3.4.6	Private -> Protected	—	—
c.3.5	Class inheritance (*Virtual/Non-virtual*)		
c.3.5.1	Virtual -> Non-virtual	—	L
c.3.5.2	Non-virtual -> Virtual	—	L
c.3.6	Class addition	—	—
c.3.7	Class inheritance structure		
c.3.7.1	Add abstract class	S+G+H+I+ *ie*(3.1.1)	L
c.3.7.2	Add non-abstract class	H	L
c.3.7.3	Delete abstract class	H + F + *ie*(3.1.2)	L
c.3.7.4	Delete non-abstract class	H+F	L

(G) Aggregation

(i) *By reference*:

```
class C2 {
    ...
    // b is a declared variable of type pointer to C1.
    C1 * b;
    // In constructor,
    C2::C2(C1& other) {
        // b is dynamically created.
        b = new C1(other);
    }
    ...
};
```

(ii) *By value*:

```
class C2 {
    ...
    // An instance of C1 is part of C2.
    C1 b;
    ...
}
```

(H) Inheritance

```
// C2 inherits from C1 in a private manner,
class C2: private C1 {...};
// in a public manner,
class C2: public C1 {...};
// in a protected manner.
class C2: protected C1{...};
```

(I) Method Invocation

```
class C2 {
    ...
    // o1 is either a global object or
    // object reference of C1.
    o1.C1_m (... );
    // o1 is a global object pointer of C1.
    o1 -> C1_m(... );
    ...
};
```

(F) Friendship

```
class C1 {
    ...
    // all methods of C2 can access
    // any member of C1.
    friend class C2;
    ...
};
```

3.3.3. Impact

The impact model predicts which classes would be impacted if a change was actually made. For practical reasons, we consider only changes which have a syntactic impact.

To calculate the impact of each identified change, a truth table is set up for that change with the five links appearing in the top (see Table 2). For each row, representing one configuration of these five links, we investigate whether there is impact (**I**) or not (**X**), and the row is marked accordingly in the 'Result' column. In some cases, it may happen that the state underlying the row cannot exist, and the row is marked with '-'. For example, when there is a change in the return type of a pure virtual method, the rows in which **G** or **I** appear cannot be investigated since neither the abstract class can be instantiated as an object (**G**) nor the pure virtual method can be invoked (**I**). For each row, the appropriate Boolean expression is derived and reduced, if possible, and the term "***L***" is appended if there is local impact. For example, for a deletion of a non-abstract class in the class inheritance structure (code change from "class c2: *c1*, c0 {... }" to "class c2: c0 {... }"), the corresponding expression is $\mathbf{H} + \mathbf{F} + \boldsymbol{L}$ which implies there is impact in derived classes (**H**), in friend classes (**F**) and locally (**L**) too. It may also happen that a change triggers another change (triggered change) to occur. For example, addition of a pure virtual method in a non-abstract class results in a triggered change since the class is now turned abstract. To indicate a triggered change, we add a note to the final expression in the form *ie* (*change id*)-'ie' stands for impact expression and 'change id' refers to the triggered change).

At this point, we wish to emphasize on the type of impact we are looking for in our work when it comes to changes in C++ systems. A given change is characterized by a transformation of the code somewhere in the system. If the system is successfully re-compiled, then there is no impact. Otherwise, we are faced with an impact, i.e., code modifications that must be done elsewhere in the system to obtain a syntactically correct code that will re-compile. Semantic issues relating to the code transformation are overlooked at this point because they cannot be inferred from the source code alone. Since our focus is only on syntactic impact of a change, the appropriate measures we have to apply are based on impact that is only dependent on the static nature of the source code. Thus, impact that may arise during run-time due to polymorphism is not catered for. Finally, we note that for some changes, it is known that there is impact (certain impact), whereas in other instances, it is clear that there is no impact at all (null impact). Yet, in some cases, it is not known whether there is impact or not (uncertain impact) until the corresponding piece of code is closely examined. For

Table 2
Truth table for change in scope of variable from public to private

Links of class C2 with class C1					*Result? Impact on class C2*
S	**G**	**H**	**I**	**F**	
Y	Y	Y	Y	Y	X
Y	Y	Y	Y	N	**I**
Y	Y	Y	N	Y	X
Y	Y	Y	N	N	**I**
Y	Y	N	Y	Y	X
Y	Y	N	Y	N	**I**
Y	Y	N	N	Y	X
Y	Y	N	N	N	**I**
Y	N	Y	Y	Y	X
Y	N	Y	Y	N	**I**
Y	N	Y	N	Y	X
Y	N	Y	N	N	**I**
Y	N	N	Y	Y	X
Y	N	N	Y	N	**I**
Y	N	N	N	Y	X
Y	N	N	N	N	**I**
N	Y	Y	Y	Y	X
N	Y	Y	Y	N	X
N	Y	Y	N	Y	X
N	Y	Y	N	N	X
N	Y	N	Y	Y	X
N	Y	N	Y	N	X
N	Y	N	N	Y	X
N	Y	N	N	N	X
N	N	Y	Y	Y	X
N	N	Y	Y	N	X
N	N	Y	N	Y	X
N	N	Y	N	N	X
N	N	N	Y	Y	X
N	N	N	Y	N	X
N	N	N	N	Y	X
N	N	N	N	N	X

example, consider the change in the return type of a pure virtual method. Derived classes may or may not define the method. If the method is not defined in a derived class, there is no impact. But, if the method is being defined in a derived class, then, there is impact in that method definition. Only by looking at the derived class definition we can determine whether there is impact or not. This type of impact (uncertain impact) has been treated as certain impact for the purpose of impact calculation.

For each of the 66 identified changes, its impact has been calculated (see Table 1). For illustration, consider the change in variable scope from public to private (code change from "*public*: int v;" to "*private*: int v;"). From the change's truth table

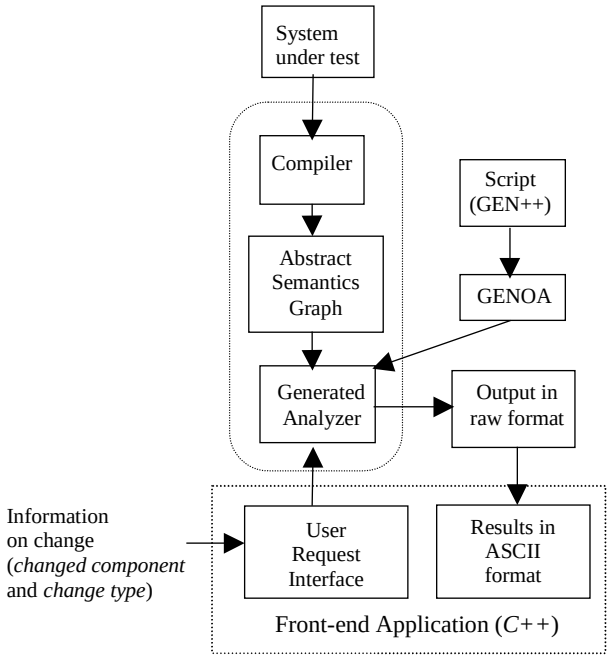

Fig. 1. Prototype of change impact model (C++)

(Table 2), we derive the *canonical expression* =

$$\mathrm{SGHI} \sim \mathrm{F} + \mathrm{SGH} \sim \mathrm{I} \sim \mathrm{F} + \mathrm{SG} \sim \mathrm{HI} \sim \mathrm{F} + \mathrm{SG} \sim \mathrm{H} \sim \mathrm{I} \sim \mathrm{F} +$$

$$\mathrm{S} \sim \mathrm{GHI} \sim \mathrm{F} + \mathrm{S} \sim \mathrm{GH} \sim \mathrm{I} \sim \mathrm{F} + \mathrm{S} \sim \mathrm{G} \sim \mathrm{HI} \sim \mathrm{F} + \mathrm{S} \sim \mathrm{G} \sim \mathrm{H} \sim \mathrm{I} \sim \mathrm{F}.$$

Reducing this expression yields $\mathbf{S} \sim \mathbf{F}$, meaning there is impact in classes which are in association ($\mathbf{S}$) with the changed class, i.e., referencing the variable, but which are not friends ($\sim \mathbf{F}$) of the changed class.

Our results show that for 25 of the 66 changes there is no impact whatsoever (neither in other classes nor locally), for four changes there is only local impact, for 37 changes there is impact in other classes, and for 11 changes there is a triggered change.

3.4. Prototype implementation

To demonstrate the practicability of our change impact model on industrial systems, we have developed the prototype system illustrated in Fig. 1. The prototype implements the model for the C++ language.

Queries are defined to calculate the impact expressions. These queries are themselves contained in scripts, i.e., high level specifications written in GEN++, the C++ implementation of GENOA [11]. Analyzers are generated from the scripts. The change type and the changed component are specified as input to a front-end application written in

C++. Once the input is validated, the front-end determines which analyzers are to be invoked, based on the type of the given change. The system under test is compiled into an abstract semantics graph, a language-independent view of the source code, using the AT&T C++ front-end. The abstract semantics graph consists of nodes which represent program elements such as expressions, statements, and the like. The analyzer runs over the abstract semantics graph gathering the information specified in the script. The output comprises the line number, class name and file name of the impact, which is sent back to the front-end application.

After performing intermediary calculations on all results received, the front-end application stores the final results in ASCII format. The front-end application can be invoked interactively or in batch mode via a shell script.

4. Change impact case study

4.1. Objectives

Using the prototype described in Section 3.4, we investigated how our model would behave on an industrial system. We hoped to learn something about the changeability of this system under test. To achieve this, we chose to determine the impact of one specific change, the signature change of a method. Descriptive statistics of the system and the impact results were then calculated. We were also interested to know whether there is any relationship between a particular OO design metric and the impact for the given change on that system.

4.2. System under test

The system under test was provided by our project partner. It is a system for decision making in telecommunications, is written in C++, and comprises 1044 classes. We extracted the C&K metrics [9] to have a first description of the system. Table 3 provides descriptive statistics of the metrics distribution. The WMC metric refers to the sum of the complexities of all the methods defined in a class. We assumed method complexity to be one for all methods, as was assumed by most authors who previously used WMC. In this case, WMC can be said to be equivalent to the number of methods defined in a class (NMC) [22].

A critical performance problem arose during the experiment. The time required to parse the system, with the prototype presented in Fig. 1, amounted to about 14 min per change impact computation for each method signature change! It was clear that with this prototype, the experiment could not be carried out on all classes of the system. Instead, we had to study samples.

The population of the 1044 system classes may be considered heterogeneous since their number of methods varies from 1 to 172! So, the stratified sampling approach [3] was applied. This caused the breaking up of the system into three groups based on

Table 3
Summary of metrics for system under test

	WMC	DIT[a]	NOC	CBO	RFC	LCOM
Min	1	0	0	0	0	0
Max	172	8	29	437	541	3587
Mean	13.50	2.87	0.88	11.71	24.10	27.61
Median	8	2	0	7	10	0
Std Dev	17.82	2.27	2.41	20.57	40.26	216.20

[a]DIT refers to the maximum path length from a class to the root class of the inheritance tree.

the WMC criterion:

- Group I (lowest WMC values) contains those classes with one or two methods.
- Classes in Group II (middle range WMC values) have not less than three and not more than 29 methods.
- The remaining classes, that is, those with at least 30 methods, were found in Group III (highest WMC values).

We ended up with 109, 837 and 98 classes in Groups I, II, and III, respectively. For Group I, the experiment was carried out on all the 109 classes. However, for Group II and III, we focussed on 30 selected classes in each group. In Group II, the selected classes were split equally among those with 12, 13, and 14 methods (classes with mean number of methods in the whole system are those with 13 methods), and in Group III, the class with the maximum number of methods was also included in the test sample.

4.3. Methodology

The primary goal of the experiment was to analyze empirically whether an OO design metric has any relationship with the impact of a change for the system under test. The change considered was the method signature change; the Boolean expression of its impact is **I**, meaning there is impact in classes where the method is invoked. The impact was calculated as the mean of the impact on every method defined in a target class. We will call this average value "change impact" of the class in the rest of this paper. The impact results were two-fold: the number of classes impacted and the number of lines impacted. We take also into account the number of impacted lines because it is widely used as a measure of maintenance effort [4,15]. The results were used in further calculations to estimate the mean and standard deviation of the change impact for the whole system. The metric chosen to correlate was the WMC[1] metric which in our experiment, is equal to the number of methods defined in a class. So, the hypothesis was laid down as

For the system under test, there is a relationship between the WMC metric and the change impact of the method signature change as defined above.

[1] For the ANOVA analysis, we assumed that WMC metrics is in ratio scaled form.

Table 4
Descriptive statistics of the impact results for the three groups

	Group I (1–2 methods)		Group II (3–29 methods)		Group III (30+ methods)	
Classes						
Total	109		837		98	
(Tested)	(109)		(30)		(30)	
Impact	**Class**	**Line**	**Class**	**Line**	**Class**	**Line**
Min	0.00	0.00	0.00	0.00	0.04	0.17
Max	19.50	125.50	3.54	11.54	7.12	8.24
Mean	**0.30**	**1.35**	**0.77**	**1.86**	**1.31**	**2.75**
Median	0.00	0.00	0.48	0.72	1.24	2.49
Std Dev	1.90	12.03	0.79	2.61	1.27	1.94

The hypothesis was first tested by the correlation analysis and, second, by the analysis of variance (ANOVA) [23]. In the ANOVA test, a null hypothesis H_0 was formulated, stating that the mean change impact is equal in all three samples. In case of rejection, the alternative hypothesis H_1 (which implies that for at least two samples, the mean change impact differs) would be accepted.

4.4. Results

4.4.1. Impact results

Descriptive statistics of the impact results for both classes and lines are summarized in Table 4. Only in Group III, there is no class with null impact. In both Groups I and III, one single class yields the maximum values for class and line impact; in Group II, however, these values result from two different classes. The mean value for class and line impact increases through Group I–III.

4.4.2. Mean and standard deviation of change impact

The results below are related to the change impact in terms of number of classes. Using the stratified sampling approach, the estimate variance of each sample (the three groups) was calculated. The three estimate variances were then used to compute the estimate mean of the whole system. With 95% confidence, the mean value of the change impact lies between 0.54 and 0.99. The standard deviation of the change impact was also estimated for the whole system by combining the three samples into one, and the value was found to be between 1.52 and 1.89.

4.4.3. Correlation coefficient

The sample correlation coefficient between the two variables, the WMC metric, and the measured change impact were calculated. The calculated correlation coefficient was found to be somewhat weak, 0.21. A scatter-plot revealed two outlier classes. Without the outliers, the correlation coefficient rises to 0.55.

Table 5
ANOVA analysis for hypothesis testing

Source of variation	*Square sum*	*Degree of freedom*	*Mean square*	F_0	*Probability value*	F_α
Between groups	25.220	2	12.610	**4.607**	0.011	3.050
Within groups	454.320	166	2.737			
Total	479.540	168				

4.4.4. ANOVA

The null hypothesis we wanted to test is H_0: $\mu_1 = \mu_2 = \mu_3$, where μ_1, μ_2 and μ_3 represent the mean values of the change impact for the three groups. The variance within each group and between different groups were calculated (see Table 5). The calculated F_0 (4.607) was found to be greater than the Fisher value, F_α (3.050).

4.5. Interpretation of results

The system has quite flat inheritance hierarchies (refer to DIT column in Table 3) and a majority of classes with a small number (less than 20) of methods. Similar observations on industrial systems have been made by other authors [5,9,18]. But there exist a few outlier classes with a large number (more than 100) of methods including the one with 172 methods. We believe this is a sign of poor OO practice, and we contend that these classes should have been further decomposed into smaller units.

The estimate mean of the change impact provides us with an interesting result. We can say that a method signature change to the system implies, on the average, not more than one class being impacted. In other words, this system can readily absorb a method signature change. However, two outlier classes are pinpointed as classes that should be of concern for the method signature change because of their high impact value.

First, from the impact results, two anomalies are noted. We found classes with 14 methods but with null impact. Those classes were closely examined and found to contain mostly virtual methods. Since we perform static analysis of the source code, invocation of these methods may have been 're-routed' elsewhere. The second anomaly is the excessive impact (7.12 and 19.50, which are well above the mean value of 0.56 for the three samples combined) associated with two classes, one of which has only two methods.

The ANOVA test confirmed that correlation result. We obtain a value (4.607) greater than the Fischer value (3.050). This means that we have to reject the null hypothesis H_0 and accept the alternative hypothesis H_1 which implies that the mean of the change impact differs for at least two samples. We conclude that there is indeed a relationship between the WMC metric and the change impact of the method signature change for the system under test.

The calculation of the estimate of the mean value of the change impact and the correlation analysis were performed again with impacted lines replacing impacted classes. Our findings based on the class impact results were confirmed by the line impact result.

5. Lessons learned

The experiment described in Section 4 was a pilot project for probing the changeability of an industrial system. The setup described in Section 3.4 was designed as a prototype for such experimentation. From an exploration point of view, the experiment was successful, yielding many lessons.

First and foremost, code size is a problem in empirical studies of software systems. In our case, the number of classes, an indicator of size, had an impact on computing time. As reported in Section 4.2, the time required to compute one change impact was so high that we could not apply the change to all classes. As a result, the statistical analysis could not be performed on the complete population but only on a sample, and the statistical methodology had to be adjusted accordingly.

The prototype system also posed some problems. Each change impact had to be programmed by a separate script that would be processed by GENOA to generate an analyzer (see Fig. 1). That analyzer would parse the whole system each time it is called upon to look up information.

Finally, when performing the post-mortem of the experiment, we realized that some methodological aspects were specific to the change considered and could not be carried over to other changes. The case in point is the definition of how to measure impact. In our case, it was decided that for each class, the impact would be the average number of impacted classes by a change to each method's signature. But, for another change such as change to a class structure, this definition might be inappropriate.

The lessons learned can be summarized as follows:

- The system under study should be parsed only once to capture and store the abstract data representation for possible retrieval later.
- The experimentation environment should be able to handle large software systems.
- For any change, what is called impact and how to calculate this impact must be defined beforehand.
- The change impact computation should require little programming effort and should be efficient even for large systems.

The recommendations above should all be implemented before performing any statistical analysis.

6. Conclusion

In this paper, we study the changeability of software systems, which remains an aspect of maintainability yet to be uncovered. The design of a system is believed to

play a determining role in the system reaction to incoming changes. We proceed by assessing the impact of changes to a system. To do so, we define an impact model first at design level and later, for systems written in the C++ language. This model is both more complete and systematic than similar work reported in the literature. For each of the possible changes identified in C++, the impact is calculated so that necessary actions may be taken to ensure a successful system compilation after change implementation. It is found for impact calculation, the various links between the classes of a system have to be considered. In fact, the impact can be expressed as a combination of these links, as detailed in [7].

An experiment was carried out on an industrial system, using a prototype of our change impact model. The number of classes and lines impacted by applying the signature change to a method were calculated. Results showed that the method signature change is a change that the system under test can absorb quite easily since on the average, at most only one class was impacted per method signature change. Another outcome of the experiment led to the establishment of a relation between WMC, a design metric, and the mean change impact (as calculated in Section 4.3) of a class. The higher the WMC value was, the higher was the mean change impact. Since the resultant correlation coefficient of 0.55 is somewhat weak, we plan to investigate the relation on further systems.

Further work is geared in two directions. The first one is to review the prototype environment, based on the lessons learned. The basis of the new approach is a source code repository, which acts as a centralized storage for the system source code information. Queries corresponding to the changes may be issued over the repository. This should result in a more flexible architecture and more efficiency in impact calculation. As a second direction, the experimental test spectrum will be widened. We expect to find results applicable to many, if not all changes. A representative set of changes will be defined, based on category (variable, method, and class), and on assumed practicality. We also intend to experiment on more than one system. One objective is to compare the impact of changes across different systems. Yet, a more ambitious one is to find changeability results applicable to a wide category of systems.

References

[1] F.B. e Abreu, Object-oriented software engineering: measuring and controlling the development process, in: Proc. 4th Int. Conf. on Software Quality, Washington DC, USA, 1994.

[2] F.B. e Abreu, Rogério Carapuça, Candidate metrics of object-oriented software within a taxonomy framework, J. Systems Software 26 (1) (1994) 87–96.

[3] S. Alalouf, D. Labelle, J. Menard, Introduction Á la Statistique Appliquée, Addison-Wesley, Reading, MA, 1990.

[4] G. Antoniol, G. Canfora, A. De Lucia, Estimating the size of changes for evolving object oriented systems: a case study, in: Proc. 6th Int. Software Metrics Symposium, Boca Raton, FL, November 1999, pp. 250–258.

[5] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Trans. Software Eng. 22 (10) (1996) 751–761.

[6] L.C. Briand, P. Devanbu, W. Melo, An investigation into coupling measures for C++, in: ICSE97, Boston, MA, 1997, pp. 412–421.

[7] M.A. Chaumun, Change impact analysis in object-oriented systems: conceptual model and application to C++, Master's Thesis, Université de Montréal, Canada, November 1998.
[8] S.R. Chidamber, C.F. Kemerer, Towards a metrics suite for object oriented design, in: Proceedings OOPSLA, Phoenix, AZ, 1991, pp. 197–211.
[9] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Software Eng. 20 (6) (1994) 476–493.
[10] S.R. Chidamber, D.P. Darcy, C.F. Kemerer, Managerial use of metrics for object-oriented software, IEEE Trans. Software Eng. 24 (8) (1998) 629–639.
[11] P.T. Devanbu, GENOA—a customizable, language- and front-end independent code analyzer, in: Proc. 14th Int. Conf. on Software Engineering, Melbourne, Australia, 1992, pp. 307–317.
[12] J. Han, Supporting impact analysis and change propagation, in software engineering environments, proceedings, in: STEP97, London, England, July 1997, pp. 172–182.
[13] P. Hsia, A. Gupta, C. Kung, J. Peng, S. Liu, A study of the effect of architecture on maintainability of object-oriented systems, in: ICSM95, Nice, France, October 1995, pp. 17–20.
[14] ISO 9126. Software Product Quality Characteristics, http://www.cse.dcu.ie/essiscope/.
[15] G.A. Kiran, S. Haripriya, P. Jalote, Effect of object orientation on maintainability of software, in: ICSM97, Bari, Italy, October 1–3, 1997, pp. 114–121.
[16] D.C. Kung, J. Gao, P. Hsia, J. Lin, Y. Toyoshima, Class firewall, test order, and regression testing of object-oriented programs, J. Object-Oriented Program. 8 (2) (1995) 51–65.
[17] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen, Change impact identification in object oriented software maintenance, in: ICSM94, Victoria, B.C., Canada, September 1994, pp. 202–211.
[18] W. Li, S. Henry, Object-oriented metrics that predict maintainability, J. Systems Software 23 (1993) 111–122.
[19] W. Li, S. Henry, D. Kafura, R. Schulman, Measuring object-oriented design, J. Object-Oriented Program. 8 (4) (1995) 48–55.
[20] L. Li, A.J. Offutt, Algorithmic analysis of the impact of changes to object-oriented software, in: ICSM96, 1996, pp. 171–184.
[21] M. Lindvall, Measurment of change: stable and change-prone constructs in a commercial C++ system, in: Proc. 6th Int. Software Metrics Symposium, Boca Raton, FL, November 1999, pp. 40–49.
[22] M. Lorenz, Object-Oriented Software Development: A Practical Guide, Prentice-Hall, Englewood Cliffs, NJ, 1993, pp. 227.
[23] J-M. Matel, R. Nadeau, Statistique en Gestion et en Économie, Gaëtan Morin Éditeur, Montréal, Canada, 1988.
[24] T.M. Pigoski, Practical Software Maintenance, Wiley, New York, 1997, pp. 384.
[25] H.D. Rombach, Design measurement: some lessons learned IEEE Software 7 (2) (1990) 17–25.
[26] J. Rumbaugh, I. Jacobson, G. Booch, The unified modeling language reference manual, Addison-Wisley, MA, USA, December 1998, pp. 550.